Programming Principles in Python (CSCI 503/490)

Modules & Packages

Dr. David Koop

Parsing Files

- Dealing with different formats, determining more meaningful data from files
- txt: text file
- csv: comma-separated values
- json: JavaScript object notation
- Jupyter also has viewers for these formats
- Look to use libraries to help possible
  - `import json`
  - `import csv`
  - `import pandas`

Writing Files: With Statement

- With statement does "enter" and "exit" handling:
- In the previous example, we need to remember to call `outf.close()`
- Using a with statement, this is done automatically:

```
- with open('huck-finn.txt', 'r') as f:
      for line in f:
          if 'Huckleberry' in line:
              print(line.strip())
```

- This is important for **writing** files!

```
- with open('output.txt', 'w') as f:
      for k, v in counts.items():
          f.write(k + ': ' + v + '\n')
```

- Without `with`, we need `f.close()`

Reading Binary Data

- Add a `'b'` to the open call to specify binary mode
  - `f = open('data.bin', 'rb')`
- You can read the data using the methods we have seen before, but…
  - …data comes back as a **byte string** (prefixed with `b`)
  - …you cannot read (all of) it nicely because it is not text-encoded!
  - `b'@\t!\xf9\xf0\x1b\x86n\x00\x00\x03HHello \xf0\x9f\x99\x82'`
  - Can create your own byte strings: `b'\x80\x00\x00\x00'`
- Use `struct` module to specify the contents of a byte
  - 8 bytes are double (float), next 4 are an int (32-bit), last 10 are char* (string)
  - `vf, vi, vs = struct.unpack('di10s')`

Writing Binary Data

- First, `struct.pack`, basically the reverse of unpack
  - `d_out = struct.pack(f'di{len(vs)}s', vf, vi, vs)`
  - Uses default endianness (little-endian)
- Need to encode strings: `vs.encode()`
  - `len(vs) # 7`
  - `len(vs.encode()) # 10 # length changes`
- Problems & Solutions:
  - Data is too big for the specified format (e.g. int): use `to/from_bytes`
  - String length (when to stop reading?): use null terminator `\0`

Command Line Interfaces (CLIs)

- Prompt:
  - $
  - 

- Commands
  - `$ cat <filename>`
  - `$ git init`
- Arguments/Flags: (options)
  - `$ python -h`
  - `$ head -n 5 <filename>`
  - `$ git branch fix-parsing-bug`

Assignment 4

- Assignment covers strings and files
- Reading & writing data to files
- Identifying and modifying strings
- Examine characters and string formatting

Scripts, Programs, and Libraries

- Often, interpreted ~ scripts and compiled code ~ programs/libraries
  - Python does compile **bytecode** for modules that are imported
- Modifying this usual definition a bit
  - Script: a one-off block of code meant to be run by itself, users **edit the code** if they wish to make changes
  - Program: code meant to be used in different situations, with **parameters** and **flags** to allow users to customize execution without editing the code
  - Library: code meant to be called from other scripts/programs
- In Python, can't always tell from the name what's expected, code can be both a library and a program

Program Execution

- Direct Unix execution of a program
  - Add the hashbang (`#!`) line as the **first line**, two approaches
  - `#!/usr/bin/python`
  - `#!/usr/bin/env python`
  - Sometimes specify `python3` to make sure we're running Python 3
  - File must be flagged as executable (`chmod a+x`) and have line endings
  - Then you can say: `$ ./filename.py arg1 ...`
- Executing the Python compiler/interpreter
  - `$ python filename.py arg1 ...`
- Same results either way

Writing CLI Programs

- Command Line Interface Guidelines
  - Accept flags/arguments
  - Human-readable output
  - Allow non-interactive use even if program can also be interactive
  - Add help/usage statements
  - Consider subcommand use for complex tools
  - Use simple, memorable name
  - …

Accepting Command-Line Parameters

- Parameters are received as a list of strings entitled `sys.argv`
- Need to `import sys` first
- `sys.argv[0]` is the name of the program as executed
  - Executing as `./hw01.py` or `hw01.py` will be passed as different strings
- `sys.argv[n]` is the nth argument
- `sys.executable` is the python executable being run

Using Parameters

- Test `len(sys.argv)` to make sure the correct number of parameters were passed
- Everything in `sys.argv` is a **string**, often need to cast arguments:
  - `my_value = int(sys.argv[1])`
- Guard against bad inputs
  - Test before using or deal with errors
  - Use `isnumeric` or catch the exception
  - Printing help/usage statement on error can help users

The main function

- Convention: create a function named `main()`
- Customary, but not required
  -
```
def main():
    print("Running the main function")
```
- Nothing happens in a script with this definition!

The main function

- Convention: create a function named `main()`
- Customary, but not required
  - `def main():`
    `print("Running the main function")`
- Nothing happens in a script with this definition!
- Need to call the function in our script!
- ```
  def main():
      print("Running the main function")
  main() # now, we're calling main
  ```
  ```

Using code as a module, too

- When we want to start a program once it's loaded, we include the line `main()` at the bottom of the code.
- Since Python evaluates the lines of the program during the import process, our current programs also run when they are imported into an interactive Python session or into another Python program.
- `import my_code # prints "Running the main function"`
- Generally, when we import a module, we **don't want it to execute**.

Knowing when the file is being used as a script

- Whenever a module is imported, Python creates a special variable in the module called `__name__` whose value is the name of the imported module.
- Example:

```
>>> import math
>>> math.__name__
'math'
```

- When Python code is run directly and not imported, the value of `__name__` is `'__main__'`.
- We can change the final lines of our programs to:

```
if __name__ == '__main__':
    main()
```

Modules and Packages

- Python allows you to import code from other files, even your own
- A **module** is a collection of definitions
- A **package** is an organized collection of modules
- Modules can be
  - a separate python file
  - a separate C library that is written to be used with Python
  - a built-in module contained in the interpreter
  - a module installed by the user (via conda or pip)
- All types use the same import syntax

[RealPython]

What is the purpose of having modules or packages?

What is the purpose of having modules or packages?

- Code reuse: makes life easier because others have written solutions to various problems
- Generally forces an organization of code that works together
- Standardizes interfaces; easier maintenance
- Encourages robustness, testing code

- This does take time so don't always create a module or package
  - If you're going to use a method once, it's not worth putting it in a module
  - If you're using the same methods over and over in (especially in different projects), a module or package makes sense

Module Contents

- Modules can contain
  - functions
  - variable (constant) declarations
  - import statements
  - class definitions
  - any other code
- Note that variable values can be changed in the module's namespace, but this doesn't affect other Python sessions.

Importing modules

- `import <module>`
- `import <module> as <another-identifier>`
- `from <module> import <identifer-list>`
- `from <module> import <identifer> as <another-identifier>, …`

- `import` imports from the top, `from … import` imports "inner" names
- Need to use the qualified names when using import (`foo.bar.mymethod`)
- as clause **renames** the imported name

Using an imported module

- Import module, and call functions with **fully qualified** name
  - ```
    import math
    math.log10(100)
    math.sqrt(196)
    ```
- Import module into current namespace and use **unqualified** name
  - ```
    from math import log10, sqrt
    log10(100)
    sqrt(196)
    ```

How does import work?

- When a module/package is imported, Python
  - Searches for the module/package
    - Sometimes this is internal
    - Otherwise, there are directory paths (environment variable `PYTHONPATH`) that python searches (accessible via `sys.path`)
  - Loads it
    - This will run the code in specified module (or `__init__.py` for a package)
  - Binds the loaded names to a namespace

[RealPython]

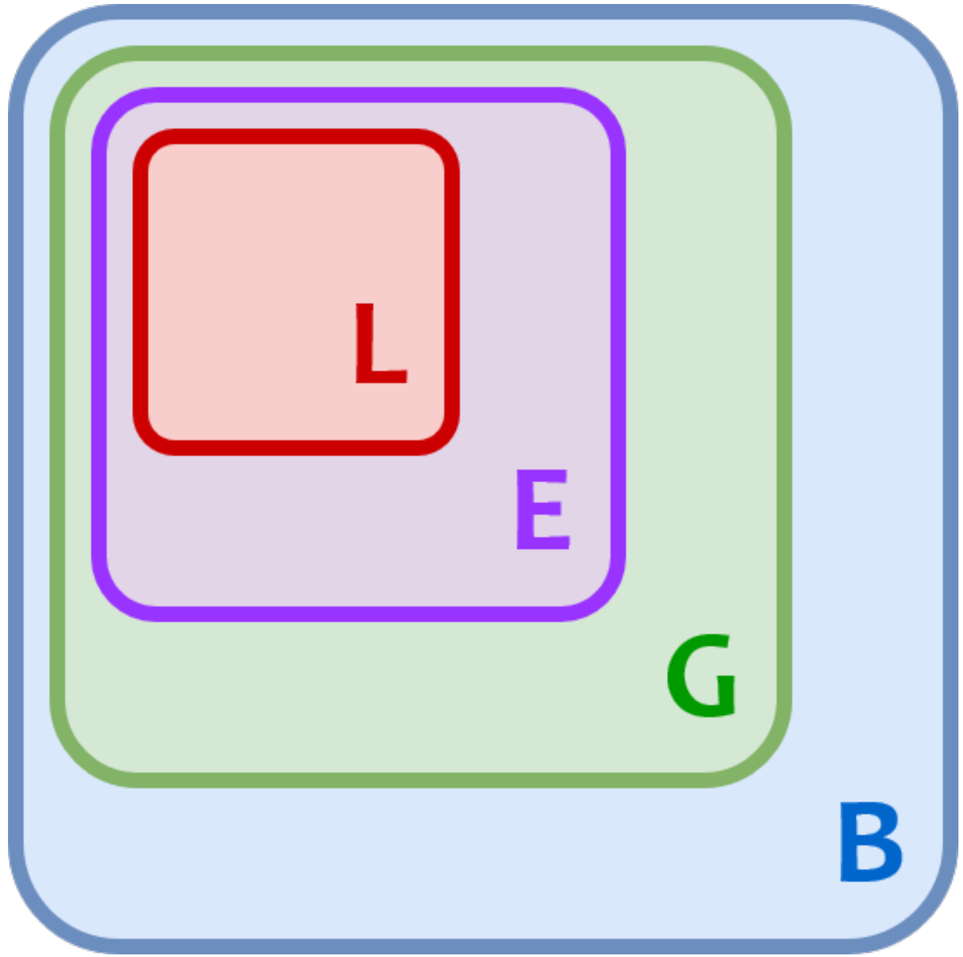
Namespaces

- An import defines a separate **namespace** while from…import adds names to the current namespace
- Four levels of namespace
  - builtins: names exposed internally in python
  - global: names defined at the outermost level (wrt functions)
  - local: names defined in the current function
  - enclosing: names defined in the outer function (when nesting functions)
-
```
def foo():
    a = 12
    def bar():
        print("This is a:", a)
```

`a` is in the **enclosing** namespace of `bar`

Namespaces

- Namespace is basically a dictionary with names and their values
- Accessing namespaces
  - `__builtins__`, `globals()`, `locals()`
- Examine contents of a namespace: `dir(<namespace>)`
- Python checks for a name in the sequence: local, enclosing, global, builtins
- To access names in outer scopes, use `global` (global) and `nonlocal` (enclosing) declarations



[RealPython]

Wildcard imports

- Wildcard imports import all names (non-private) in the module
- What about
  - `from math import *`
- Avoid this!
  - Unclear which names are available!
  - Confuses someone reading your code
  - Think about packages that define the same names!
- Allowed if republishing internal interface (e.g. in a package, you're exposing functions defined in different modules

Import Guidelines (from PEP 8)

- Imports should be on separate lines
  - ~~`import sys, os`~~
  -
```
import sys
import os
```
- When importing multiple names from the same package, do use same line
  - `from subprocess import Popen, PIPE`
- Imports should be at the **top** of the file (order: standard, third-party, local)
- Avoid wildcard imports in most cases

Conditional or Dynamic Imports

- Best practice is to put all imports at the beginning of the py file
- Sometimes, a conditional import is required
  -
```
if sys.version_info >= [3,7]:
    OrderedDict = dict
else:
    from collections import OrderedDict
```
- Can also dynamically load a module
  - `import importlib`
  - `importlib.import_module("collections")`
  - The `__import__` method can also be used